Verifying Bit-Manipulations of Floating-Point

Wonyeol Lee Rahul Sharma Alex Aiken
Stanford University, USA
{wonyeol, sharmar, aiken}@cs.stanford.edu

Abstract

Reasoning about floating-point is difficult and becomes only more so if there is an interplay between floating-point and bit-level operations. Even though real-world floating-point libraries use implementations that have such *mixed* computations, no systematic technique to verify the correctness of the implementations of such computations is known. In this paper, we present the first general technique for verifying the correctness of mixed binaries, which combines abstraction, analytical optimization, and testing. The technique provides a method to compute an error bound of a given implementation with respect to its mathematical specification. We apply our technique to Intel's implementations of transcendental functions and prove formal error bounds for these widely used routines.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs

Keywords Verification; Floating-point; Bit-manipulation; Bit-level operation; ULP error; Absolute error; x86 binary; Binary analysis; Transcendental function

1. Introduction

Highly optimized implementations of floating-point libraries rely on intermixing floating-point and bit-level code. Even though such code is part of widely used libraries, such as optimized implementations of C math libraries, automatic formal verification of these implementations has remained an open challenge [22]. Although it has been demonstrated that it is possible to construct machine-checkable proofs of correctness by hand for floating-point algorithms of the level of sophistication we are interested in [11, 12], no existing automated verification technique is capable of analyzing these implementations. In this paper, we present a first step towards addressing this challenge.

Bit-precise floating-point reasoning is hard: floating-point is an approximation to real arithmetic, but floating-point numbers do not obey the algebraic axioms of real numbers due to rounding errors. The situation becomes even more difficult in the presence of bit-level operations, such as bit-manipulations of the floating-point representation. To illustrate a *mixed* code, consider an implementation that computes the floating-point number 2^n from a small integer n. A naïve implementation would first compute the integer representing 2^n and then perform the computationally expensive operation of converting an integer to a floating-point number. Alternatively, the same result can be obtained by bit-shifting $n + 1023$ left by 52 bits (Figure 3). Existing static analyses for floating-point arithmetic would be stumped by the bit-shift operation and would fail to prove the functional correctness of this trick. Moreover, such tricks are routine in real codes [9, 16].

Before explaining our solution, it is important to understand why existing automated techniques (based on testing, model checking, and abstract interpretation) are inadequate. The simplest verification technique is exhaustive testing of all possible inputs. This approach is feasible for a function like `expf` that computes the exponential of a 32-bit single precision floating-point number. However, the number of double precision floating-point numbers is too large for brute force enumeration to be tractable.

A plausible verification strategy involves encoding correctness as the validity of a SMT formula [5]. However, the specifications of interest here are transcendentals and these $(e^x, \sin(x)$, etc.) cannot be encoded precisely in existing SMT theories. Verifiers based on abstract interpretation, such as ASTRÉE and FLUCTUAT, use pattern matching to handle certain bit-trick routines in commercial floating-point avionics codes [9, 16]. Our goal is a general technique.

Our approach to the problem is to divide and conquer. For a given floating-point implementation, we consider non-overlapping intervals that are subsets of the possible range of inputs. We require each interval I to satisfy the following property: if we statically know that the inputs are restricted to I, the bit-level operations can be removed from the im-

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908107

plementation by partial evaluation. Then, for each interval, we have a specialized implementation that is composed exclusively of floating-point operations and thus amenable to abstraction-based techniques. Our main contribution is to devise a procedure to construct such intervals (§4). There is one significant subtlety: The intervals do not always fully cover the space and we must deal with potential "gaps" between intervals. Commercial tools such as FLUCTUAT [7, 9] also subdivide the input range (with no gaps) to improve precision and our technique can be seen as a systematic method to construct these subdivisions. We analyze the implementations specialized for each interval and report the maximum error between the implementation and the ideal mathematical specification.

We make the following contributions.

- We describe the first general technique for verification of mixed floating-point and bit-level code. We are unaware of any automatic or semi-automatic verification technique that can prove the functional correctness of the production grade benchmarks we consider. Prior to this work, formal verification of such benchmarks required manual construction of machine-checkable proofs [11, 12].
- We reduce the problem of computing bounds on numerical errors to an optimization problem and leverage state-of-the-art techniques for analytical optimization. While our method is not fully automatic, these techniques automate one of the most difficult aspects of the problem and make verification of complex implementations feasible.
- Our technique performs verification at the binary level, not on source code or a model of the program. Thus, the derived bounds apply to the actual code that executes directly on the hardware.

We evaluate our technique on three implementations of transcendental functions from Intel's libraries: a bounded periodic function (`sin`, §5.2), an unbounded discontinuous periodic function (`tan`, §5.3), and an unbounded continuous function (`log`, §5.4). We are able to successfully bound the difference between the result computed by these implementations and the exact mathematical result. For each of these functions, we also trade precision for performance and create significantly more efficient variants that produce approximately correct results. Using our technique, we are able to provide a bound on the difference between the approximate variants and the mathematical specifications. These results demonstrate the generality of our technique and address some of the drawbacks of manually constructed proofs: modifying the manual proofs to prove even slightly different theorems is difficult [11, 12]. To quote Harrison [11],

> *[N]ontrivial proofs, as are carried out in the work described here, often require long and complicated sequence of rules. The construction of these proofs often requires considerable persistence. Moreover, the resulting proof scripts can be quite hard to read, and in some cases hard to modify to prove a slightly different theorem.*

```
vmovddup   %xmm0, %xmm0
vmulpd     L2E,   %xmm0, %xmm2
vroundpd   $0,    %xmm2, %xmm2
--------------------------------
vcvtpd2dqx %xmm2, %xmm3
vpaddd     B,     %xmm3, %xmm3
vpslld     $20,   %xmm3, %xmm3
vpshufd    $114,  %xmm3, %xmm3
--------------------------------
vmulpd     C1,    %xmm2, %xmm1
vmulpd     C2,    %xmm2, %xmm2
vaddpd     %xmm1, %xmm0, %xmm1
vaddpd     %xmm2, %xmm1, %xmm1
--------------------------------
vmovapd    T1,    %xmm0
vmulpd     T12,   %xmm1, %xmm2
vaddpd     T11,   %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T10,   %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T9,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T8,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T7,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T6,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T5,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T4,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T3,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     T2,    %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     %xmm0, %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm1
vaddpd     %xmm0, %xmm1, %xmm0
--------------------------------
vmulpd     %xmm3, %xmm0, %xmm0
retq
```

Figure 1. The x86 assembly code of `exp` that ships with S3D [3]. Instructions have been reordered to aid understanding, without affecting the output.

The rest of the paper is organized as follows. §2, through an example, discusses our verification technique. §3 reviews formal definitions of rounding errors and §4 presents our verification technique that combines abstraction, analytical optimization, and testing. §5 discusses evaluation and §6 surveys prior work. Finally, §7 gives a discussion of future work and §8 concludes.

2. Motivating Example

S3D [3] is a combustion chemistry simulation that is heavily used in research on developing more efficient and cleaner fuels for internal combustion engines. The performance of the exponential function is so important for this task that the developers ship a hand-coded x86 assembly implementation

```
 1  vmulpd      L2E,    %xmm0, %xmm2
 2  vroundpd    $0xfffffffffffffffe, %xmm2, %xmm2
 3  vcvttpd2dq  %xmm2,  %xmm3
 4  vpaddw      B,      %xmm3, %xmm3
 5  vpsllq      $0x14,  %xmm3, %xmm3
 6  vpshufd     $0x3,   %xmm3, %xmm3
 7  vmulpd      C1,     %xmm2, %xmm1
 8  vaddpd      %xmm1,  %xmm0, %xmm1
 9  vmovapd     T1,     %xmm0
10  vlddqu      T8,     %xmm2
11  vmulpd      %xmm1,  %xmm2, %xmm2
12  vaddpd      T7,     %xmm2, %xmm2
13  vmulpd      %xmm1,  %xmm2, %xmm2
14  vaddpd      T6,     %xmm2, %xmm2
15  vmulsd      %xmm1,  %xmm2, %xmm2
16  vaddpd      T5,     %xmm2, %xmm2
17  vmulpd      %xmm1,  %xmm2, %xmm2
18  vaddpd      T4,     %xmm2, %xmm2
19  vmulpd      %xmm1,  %xmm2, %xmm2
20  vaddpd      T3,     %xmm2, %xmm2
21  vmulsd      %xmm1,  %xmm2, %xmm2
22  vaddpd      T2,     %xmm2, %xmm2
23  vmulsd      %xmm1,  %xmm2, %xmm2
24  vaddsd      %xmm0,  %xmm2, %xmm2
25  vmulpd      %xmm1,  %xmm2, %xmm1
26  vaddsd      %xmm0,  %xmm1, %xmm0
27  vmulpd      %xmm3,  %xmm0, %xmm0
28  retq
```

Figure 2. The x86 assembly code of $\texttt{exp}_{\texttt{opt}}$ automatically generated by STOKE [22]. This code is less precise but has better performance compared to `exp` (Figure 1).

`exp` (Figure 1), which is inspired by the implementation of the exponential function present in CUDA libraries for GPUs. There is no source code as it has been implemented directly in assembly. There is also no documentation available regarding `exp` except that it is supposed to compute e^x for $x \in [-2.6, 0.12]$. As is characteristic of highly optimized floating-point implementations, `exp` contains bit-level operations (rounding to integer on line 3, converting double to integer on line 4, bit-vector addition on line 5, bit-shift on line 6, and bit-shuffle on line 7).

The main algorithm used by `exp` first computes an integer N and a reduced value d:

$$N = \text{round}\left(x \cdot \log_2 e\right), \ d = x - (\log 2)N, \qquad (1)$$

where $\log(\cdot)$ without a base denotes the natural logarithm. Then it uses the following identity to compute e^x:

$$e^x = e^{(\log 2)N} \cdot e^d = 2^N \cdot e^d$$

The implementation `exp` computes 2^N (a double in IEEE representation) from N (a 64-bit integer in 2's complement representation) using bit-level operations. It computes e^d using a Taylor series expansion of degree 12:

$$e^d \approx \sum_{i=0}^{12} \frac{d^i}{i!}$$

Next, we relate this algorithm with Figure 1. Our description below elides several details that are important for performance (such as the use of vector instructions), and focuses on functionality. The calling convention used by `exp` includes storing the first argument and the return value of a function in the register `xmm0`. We omit details about the x86 syntax and describe the implementation at a high level. The code is divided into several blocks by the horizontal lines in Figure 1 and the instructions within a block compute a value of interest.

- The first block (lines 1-3) computes N from the input x. The second instruction multiplies x by `L2E` (the double closest to $\log_2 e$) and the third instruction rounds the result.
- The second block (lines 4-7) computes 2^N from N, using bit-vector addition (line 5), bit-shift (line 6), and bit-shuffle (line 7). The bit-vector represented by `B` is `0x000003ff000003ff` (in hex). Recall that `0x3ff` in hex is 1023 in decimal, which is the bias used for the exponent in doubles (§3).
- The third group (lines 8-11) computes d from x and N. This computation uses $\log 2$ which is a transcendental number (Equation 1). To maintain accuracy, $\log 2$ is represented as the sum of two doubles: $c_1 = -0.69315\cdots$ (line 8) and $c_2 = -2.31904\cdots \times 10^{-17}$ (line 9) where $\log 2 \approx -c_1 - c_2$. This representation effectively provides more than a hundred bits to represent $\log 2$ with the desired accuracy. Using c_1 and c_2, we can compute $d \approx (x + c_1 N) + c_2 N$ (lines 10-11).
- The fourth block (lines 12-36) computes e^d from d, using a Taylor series expansion of degree 12. The constant $\texttt{T}i$ represents $\frac{1}{i!}$.
- The last group (line 37-38) returns $e^x = 2^N \cdot e^d$. Recall that 2^N is computed exactly by the second block and e^d is computed approximately by the fourth block.

Given this non-trivial implementation, a valid question is: What does it achieve? Fundamentally, since there are only a finite number of floating-point numbers, no implementation can compute e^x exactly. All floating-point implementations provide only an approximate result which has finite precision. Therefore, one possible measure of a correctness of such implementations is given by *precision loss*: the deviation of the computed answer from the mathematically exact result. Our goal in this paper is to provide sound bounds on the maximum precision loss of such implementations.

The first challenge associated with floating-point implementations is that of rounding errors. As is standard, we model rounding error using non-determinism. For example, line 2 of Figure 1 multiplies the input x and a constant `L2E`, and we model its output as an element chosen non-deterministically from the set $\{(x \times \texttt{L2E})(1+\delta) : |\delta| < \epsilon\}$, where $+$ and $\times$ denote real-number addition and multiplication, respectively. The quantity δ models the rounding error and the *machine epsilon* ϵ provides a bound on the rounding errors of floating-point multiplication (§3).

The next challenge, which has not been systematically addressed previously, is associated with bit-level operations. We use a divide-and-conquer approach to address the challenge. We denote the set of possible inputs by the interval X and create a set of intervals $\mathcal{I} = \{I_k : k \in \mathbb{Z} \text{ and } I_k \subseteq X\}$ such that the following property holds:

$$\forall x \in I_k.\ \{(x \times \mathtt{L2E})(1+\delta) : |\delta| < \epsilon\} \subset \left(k - \frac{1}{2}, k + \frac{1}{2}\right) \quad (2)$$

This decomposition is useful because it ensures that for each input $x \in I_k$ the rounded output N of line 3 is k. We show how to obtain this decomposition in §4.

Given such a decomposition $\mathcal{I}$, we run $|\mathcal{I}|$ separate analyses, where the k^{th} analysis restricts the inputs to I_k. In the k^{th} analysis, the result N of rounding (line 3) is always k which is the only input to the bit-level operations of the second block (lines 4-7). Hence, it is sound to replace the bit-level operations by an instruction that moves the double 2^k to the register `xmm3`. This specialized code consists exclusively of floating-point operations.

Next, each analysis generates a separate *symbolic representation* that over-approximates the possible outputs of its specialized code. The symbolic representation $\mathcal{A}_{\vec{\delta}}(x)$ is a function of the input x and the rounding errors $\vec{\delta}$. For example, the symbolic representation of the multiplication of x and `L2E` (line 2) is given by $(x \times \mathtt{L2E})(1+\delta)$. In general, if an expression e_1 has symbolic representation $\mathcal{A}'_{\vec{\delta}}(x)$ and e_2 has symbolic representation $\mathcal{A}''_{\vec{\delta}}(x)$ then the symbolic representation of $e_1 \otimes_{\mathtt{f}} e_2$ (where $\otimes_{\mathtt{f}}$ is floating-point addition or multiplication) is given by $(\mathcal{A}'_{\vec{\delta}}(x) \otimes \mathcal{A}''_{\vec{\delta}}(x))(1+\delta')$, where $\otimes$ is real-number addition or multiplication and δ' is a new variable representing the rounding error of the operation $\otimes_{\mathtt{f}}$.

Finally, each analysis uses analytical optimization to maximize the difference between the symbolic representation and the ideal mathematical result in the interval of interest. Several representations of precision loss are possible. If we measure precision loss using *absolute error* (§3) then the k^{th} analysis solves the following optimization problem:

$$\max_{x \in I_k, \vec{\delta}} \left| \mathcal{A}_{\vec{\delta}}(x) - e^x \right|$$

For `exp`, we use the set of inputs $X = [-4, 4]$ that results in 13 intervals, i.e., $|\mathcal{I}| = 13$ and the symbolic representations have a maximum of 29 δ variables for rounding errors. The maximum absolute error reported across all intervals is 5.6×10^{-14}.

There is one remaining issue. The division of the entire input range X into intervals $\mathcal{I} = \{I_k\}$ may result in some $x \in X$ where $x \notin I_k$ for any k. For example, consider the input $\frac{1}{2 \cdot \mathtt{L2E}}$. Due to rounding errors, it is not clear whether, for this input, the result N of line 3 would be 0 or 1. Therefore, this input is not included in any $I_k \in \mathcal{I}$ (Equation 2). For `exp`, there are 36 such floating-point numbers that are not included in any interval. For these inputs, we simply execute the program on each one and directly measure the precision loss. The maximum absolute error for these 36 inputs is 2.1×10^{-14} (which is close to but less than 5.6×10^{-14}).

Another common representation for precision loss is ULP error (§3) that signifies the number of floating-point numbers between the computed and the mathematically exact result. We show how to compute ULP error (for each interval) in §4.4. For `exp`, the maximum ULP error over X is 14, that is, there are at most 14 floating-point numbers between the output of `exp` and e^x. We are unaware of any previous automated analysis that can prove this result.

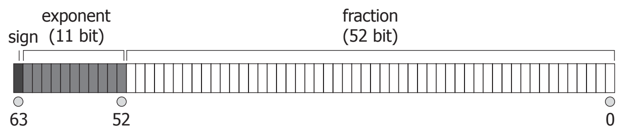
It is possible to further improve the performance of `exp` by sacrificing more precision. Despite its heavy use of the exponential function, S3D loses precision elsewhere and does not require precise results from `exp` to maintain correctness. One possible strategy involves asking a developer to create a customized implementation that has better performance at the cost of less precise results (§5). Such implementations can also be automatically generated using stochastic optimizers such as STOKE [22]. At a high level, STOKE makes random changes to binaries until they get faster and remain "approximately correct" on some tests.

The binary $\mathtt{exp}_{\mathtt{opt}}$ (Figure 2) is automatically generated by making random changes to `exp` using STOKE. The differences between $\mathtt{exp}_{\mathtt{opt}}$ and `exp` are that $\mathtt{exp}_{\mathtt{opt}}$ computes d as $x + c_1 N$ (without using c_2) and it uses a Taylor series expansion of degree 8 (instead of 12). The authors of [22] claim that $\mathtt{exp}_{\mathtt{opt}}$ has an ULP error below 10^7 from `exp` and improves the performance of the diffusion task of S3D by 27%. However, the correctness guarantees provided by STOKE are statistical and there is no formal guarantee that $\mathtt{exp}_{\mathtt{opt}}$ cannot produce results with an ULP error much greater than 10^7 for some unobserved input.

We use our analysis to bound the precision loss between `exp` and $\mathtt{exp}_{\mathtt{opt}}$. Our analysis reports that the maximum absolute error between `exp` and $\mathtt{exp}_{\mathtt{opt}}$ is 1.2×10^{-8} and the maximum ULP error is 1.9×10^6 which though large is well below the desired bound of 10^7 ULPs. Therefore, we have proven formally that the STOKE generated code respects the ULP bound of 10^7 for all inputs $x \in [-4, 4]$. This verification task was left as a challenge in [22].

3. Preliminaries

Figure 3 describes the bit representation of 64-bit double precision floating-point numbers according to the IEEE 754 standard. The most significant bit is the sign bit (denoted by s) which determines whether the number is positive or negative. The next 11 bits represent the exponent (denoted by p) and the remaining 52 bits represent the significand or the fractional part (denoted by f). Figure 3 shows the values represented by different bit patterns. Unless otherwise stated, a *floating-point number* is such a double precision



Type	Exponent (p)	Fraction (f)	Value
Zero	0	0	$(-1)^s \cdot 0$
Denormal	0	$\neq 0$	$(-1)^s \cdot 2^{-1022} \cdot 0.f$
Normal	$[1, 2046]$	unconstrained	$(-1)^s \cdot 2^{p-1023} \cdot 1.f$
Infinity	2047	0	$(-1)^s \cdot \infty$
NaN	2047	$\neq 0$	$(-1)^s \cdot \perp$

Figure 3. IEEE-754 double precision floating-point format.

floating-point number. For simplicity we do not consider denormals, but it is straightforward to incorporate them in our verification techniques (see the footnote 4 in §4.3).

The result of applying an arithmetic operation to floating-point number(s) need not to be expressible as a floating-point number. In this case, the result r is rounded to the closest representable floating-point number, denoted by $\mathrm{fl}(r)$. The IEEE standard provides algorithms for addition, subtraction, multiplication, and division, and requires the implementations following the standard to produce exactly the results specified by the algorithms. If an architecture obeys the IEEE standard then we have the following guarantee[1]:

$$x \otimes_{\mathsf{f}} y = \mathrm{fl}(x \otimes y) \in \left\{(x \otimes y)(1+\delta) : |\delta| < 2^{-53}\right\} \quad (3)$$

That is, if we have two floating-point numbers x and y and apply a floating-point operation $\otimes_{\mathsf{f}}$ included in the IEEE standard, then the result is the closest floating-point number to the real number obtained by treating x and y as real numbers and applying the exact operation $\otimes$ over the reals. The floating-point result is guaranteed to belong to a very small interval around the actual result. We call the constant 2^{-53} *machine epsilon*[2] and denote it by ϵ.

There are three standard methods for representing rounding errors. The *absolute error* simply measures the magnitude of the difference between the two results:

$$abs\_err(r_1, r_2) = |r_1 - r_2|$$

For example, in Equation 3 the maximum absolute error between the computed result and the actual result is $|(x \otimes y)\epsilon|$. Therefore, this error grows with the magnitude of $(x \otimes y)$. In contrast, the *relative error* is the following:

$$rel\_err(r_1, r_2) = \left|\frac{r_1 - r_2}{r_2}\right|$$

[1] Equation 3 is valid because we neglect underflows, overflows, and denormals. In general, if $|x \otimes y| < 2^{1024}$ then it is always true that $\mathrm{fl}(x \otimes y) \in \{(x \otimes y)(1+\delta) + \delta' : |\delta| < 2^{-53}, |\delta'| < 2^{-1075}\}$ [18].

[2] The IEEE standard does not define machine epsilon and different definitions can be found in the literature. Regardless, Equation 3 is valid.

For Equation 3, the maximum relative error is ϵ which is independent of x and y. The relative error is traditionally expressed as a multiple of ϵ.

The final representation is "units in last place" error or *ULP error*. If $d = 1.f_1 \ldots f_{52} \times 2^p$ is used to represent a real number r then it is in error by

$$\mathrm{ULP}(d, r) = \left|1.f_1 \ldots f_{52} - \frac{r}{2^p}\right| 2^{52}$$

units in last place. For Equation 3, the maximum ULP error is $\frac{1}{2}$, which is attained when r lies exactly in between two consecutive floating-point numbers. The relative error expressed as a multiple of machine epsilon and the ULP error are guaranteed to be within a factor of two for floating-point numbers [8]. Intel claims[3] that the maximum error of its implementations of transcendentals is within 1 ULP, which is typical for well-engineered implementations of transcendental functions. Therefore, if r represents the mathematically exact result and d is the floating-point number closest to r then the belief is that the implementation produces a result which is d, or d_- (the floating-point number just below d), or d_+ (the floating-point number just above d).

Remark. Note that almost half of all floating-point numbers lie in the interval $[-1, 1]$ and the density of floating-point numbers increases almost exponentially as we approach 0. Hence, for floating-point numbers x and y that are near 0, even if the absolute error $|x - y|$ is small, the ULP error $\mathrm{ULP}(x, y)$ can still be huge. For instance, if x is zero and $|x - y| \approx \epsilon$ then $\mathrm{ULP}(x, y) \approx 4 \times 10^{18}$. Thus all analysis techniques based on propagating round-off errors, including ours, must necessarily yield large ULP errors for many floating-point operations that produce results very close to 0.

4. Formalism

We now describe our procedure for estimating the precision loss of an implementation with respect to its mathematical specification. The procedure has two main steps. First, we construct a symbolic abstraction that soundly over-approximates the implementation. Next, we use an analytical optimization procedure to obtain a precise bound on the deviations the symbolic abstraction can have from the mathematical specification. We start by defining the syntax of a core language that we use for the formal development.

4.1 Core Language

Recall that the implementations of interest are highly optimized x86 binaries. The x86 ISA has more than two thousand different instruction variants, so we desugar x86 programs into a core expression language that is the subject of our analysis.

Figure 4 shows the grammar representing the expressions in this language. An *expression* e in the core language can

[3] `https://software.intel.com/sites/default/files/article/326703/fp-control-2012-08.pdf`

$$\begin{array}{rccl}
\text{expression} & e & ::= & c \mid \chi \mid \mathsf{L}[e] \mid \\
 & & & e \otimes_{\mathsf{b}} e \mid e \otimes_{\mathsf{s}} e \mid \\
 & & & e \otimes_{\mathsf{i}} e \mid e \otimes_{\mathsf{f}} e \mid \ \otimes_{\mathsf{c}} e \\
\text{bitwise operators} & \otimes_{\mathsf{b}} & \in & \{\mathsf{AND}, \mathsf{OR}, \cdots\} \\
\text{shift operators} & \otimes_{\mathsf{s}} & \in & \{\ll, \gg, \cdots\} \\
\text{bit-vector arithmetic operators} & \otimes_{\mathsf{i}} & \in & \{+_{\mathsf{i}}, -_{\mathsf{i}}, \times_{\mathsf{i}}, \cdots\} \\
\text{floating-point operators} & \otimes_{\mathsf{f}} & \in & \{+_{\mathsf{f}}, \times_{\mathsf{f}}, /_{\mathsf{f}}, \cdots\} \\
\text{casting operators} & \otimes_{\mathsf{c}} & \in & \{\mathsf{i2f}, \mathsf{f2i}, \mathsf{round}, \cdots\}
\end{array}$$

Figure 4. The syntax of the core language.

be a 64-bit constant c, a 64-bit input χ, the result of a table lookup $\mathsf{L}[\cdot]$, or the application of a unary or a binary operator to subexpression(s). All expressions in this language evaluate to 64-bit values. The x86 implementations often use 128-bit SSE registers and SIMD instructions for efficiency. However, these operations can be expressed in our language by desugaring them to multiple operations applied to multiple 64-bit expressions. For brevity, we also restrict our presentation to settings with only a single input and a single lookup table. It is straightforward to generalize our results to implementations with multiple inputs and multiple tables (but see discussion in §7 of scaling issues in handling multiple inputs).

The operators relevant for our benchmarks are shown in Figure 4. The bitwise operators include bitwise-and (AND) and bitwise-or (OR) that are used for masking bits. The left shift ($\ll$) and the right shift ($\gg$) operators are "logical" shifts (as opposed to "arithmetic" shifts) and introduce zeros. The bit-vector arithmetic operators are "signed" operators that interpret the argument bits as 64-bit integers in the 2's complement notation. The floating-point operators interpret the argument bits as 64-bit IEEE 754 double precision floating-point numbers. The casting operator i2f consumes a bit-string, interprets it as an integer written in the 2's complement notation (e.g., 42), and generates a bit-string that when interpreted as a floating-point number represents a value equal to the integer (e.g., 42.0). The round operator rounds to the nearest integer (e.g, 42.1 is rounded to 42.0) and the f2i operator first rounds and then converts the rounded floating-point number to a 64-bit integer.

For any expression e, the concrete semantics is denoted by $\mathcal{E}(e) : \{0,1\}^{64} \to \{0,1\}^{64}$. That is, the value obtained by evaluating e with an input x is given by $\mathcal{E}(e)(x)$. The definition of $\mathcal{E}(\cdot)$ is standard and we omit it.

4.2 Symbolic Abstractions

Our goal is to compute a symbolic representation that over-approximates the behaviors of an expression e. Constructing this abstraction is difficult due to the interplay between floating-point and *bit-level* (bitwise, shift, bit-vector arithmetic, and casting) operations. Therefore, we define this abstraction piecewise. We restrict the inputs to small enough intervals—thus ensuring that the bit-level operations are amenable to static analysis—and we construct multiple abstractions, one for each interval.

The description of our abstractions requires some operators relating real numbers and floating-point numbers. The function $\mathrm{d2R} : \{0,1\}^{64} \to \mathbb{R} \cup \{\pm\infty, \mathrm{NaN}\}$ is the natural map from 64-bit floating-point numbers to real numbers. It handles infinity and not-a-number by mapping them to $\{\pm\infty, \mathrm{NaN}\}$. This mapping is also extended to sets of floating-point numbers in the obvious way. If $\mathcal{P}(\cdot)$ denotes the power set of a given set then the inverse map $\mathrm{R2d} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\{0,1\}^{64})$ maps a subset S of real numbers to the largest set of floating-point numbers such that

$$\forall x \in \mathrm{R2d}(S).\ \mathrm{d2R}(x) \in S$$

For brevity, we use the abbreviation $\widehat{S} \triangleq \mathrm{R2d}(S)$. We use X to denote the interval over real numbers over which we want to compute the precision loss. Therefore, the input χ ranges over the set $\widehat{X} = \mathrm{R2d}(X)$.

We next define a symbolic abstraction. Let $\mathcal{I} = \{[l_1, r_1], \cdots, [l_n, r_n]\}$ denote a set of intervals in $\mathbb{R}$, where $[l_i, r_i] \subseteq X$ for all i. The *symbolic representation* $\mathcal{A}_{\vec{\delta}} : \mathbb{R} \to \mathbb{R} \cup \{\bot\}$ is a function of $x \in \mathbb{R}$ and $\vec{\delta} = (\delta_1, \cdots, \delta_m)$, where each $\delta_i \in \mathbb{R}$ represents a rounding error. The fact that $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ is an abstraction of e is defined as follows.

Definition 1. $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ *is* a symbolic abstraction *of e if for all intervals $I \in \mathcal{I}$, for all floating-point numbers $x \in \widehat{I}$,*

$$\mathrm{d2R}(\mathcal{E}(e)(x)) \in \mathcal{A}(\mathrm{d2R}(x))$$

where $\mathcal{A}(y) \triangleq [\min_{\vec{\delta}} \mathcal{A}_{\vec{\delta}}(y), \max_{\vec{\delta}} \mathcal{A}_{\vec{\delta}}(y)]$.

We discuss a procedure to construct a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of an expression e next.

4.3 Construction of Symbolic Abstractions

The expressions consist of floating-point and bit-level parts. To reason about bit-level operations, we need to keep track of subexpressions that are constant or are determined by a small subset of the bits of the input χ. To this end, we define a *book-keeping map* $\mathcal{B} : \{0,1\}^{64} \to \{0,1\}^{64} \cup \{\bot\}$. If $\mathcal{B}(x) \neq \bot$ then for all $i = 0, \cdots, 63$, the i^{th} bit of $\mathcal{B}(x)$, denoted by $\mathcal{B}(x)[i]$, is either 0, or 1, or a boolean function of the bits $x[63], \cdots, x[0]$. For instance, consider an expression $e = (x\ \mathsf{AND}\ \mathtt{0x80}^{15})\ \mathsf{OR}\ \mathtt{0x3fe0}^{13}$ which computes a floating-point number $\mathrm{sgn}(x) \times 0.5$, where b^i is the string with i b's, and $\mathrm{sgn}(\cdot)$ is the sign function. The book-keeping map for e is $\mathcal{B}(x) = x[63]01^9 0^{53}$, and it represents that the sign bit of e is that of x and the rest of the bits of e are the same as that of $\mathtt{0x3fe0}^{13}$.

Just like symbolic abstractions, the book-keeping map is also defined piecewise over different intervals of $\mathcal{I}$. We give the formal definitions of defined-ness next.

Definition 2.

$$\begin{array}{rcl}
(\mathcal{I}, \mathcal{A}_{\vec{\delta}})\ \mathsf{defined} & \Leftrightarrow & \forall I \in \mathcal{I}.\ \forall x \in I.\ \mathcal{A}_{\vec{\delta}}(x) \neq \bot \\
(\mathcal{I}, \mathcal{B})\ \mathsf{defined} & \Leftrightarrow & \forall I \in \mathcal{I}.\ \forall x \in \widehat{I}.\ \mathcal{B}(x) \neq \bot
\end{array}$$

Next, we relate these abstractions with concrete semantics.

Definition 3. $(\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$ *is* consistent *with* e *if the following hold:*

$$(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \text{ defined } \Rightarrow (\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \textit{ is a symbolic abstraction of } e,$$
$$(\mathcal{I}, \mathcal{B}) \text{ defined } \Rightarrow \forall I \in \mathcal{I}.\, \forall x \in \widehat{I}.\, \mathcal{E}(e)(x) = \mathcal{B}(x)$$

An important case is when every bit of $\mathcal{B}$ is a constant over each interval and we define it separately.

Definition 4.

$$(\mathcal{I}, \mathcal{B}) \text{ constant} \Leftrightarrow \forall I \in \mathcal{I}.\, \exists n \in \{0,1\}^{64}.\, \forall x \in \widehat{I}.\, \mathcal{B}(x) = n$$

Figure 5 lists the rules for construction of symbolic abstractions. These rules have the following form: $e \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$, read as e is provably consistent with $(\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$. These rules use the following notations. For a set S, $\mathrm{Id}_S : S \to S$ denotes the identity function. For a function $f : T \to U$ and a subset $S \subset T$, $f\restriction_S : S \to U$ is a restriction of f on S.

The rules for atomic expressions are direct (CONST and INPUT). For a table lookup, the index should be a constant over each interval (LOOKUP). Here, the result of the lookup can be obtained via the concrete semantics. For example, consider a table lookup $\mathsf{L}[e]$ where e represents the bits of the exponent of χ. Also assume that $\mathcal{I} = \{I_k : \forall x \in \widehat{I_k}.\text{ exponent of } x = k\}$. Then, the lookup corresponding to the k^{th} interval provides the bits $\mathsf{L}[k]$, which are recorded in the book-keeping map and the symbolic representation. The rule for i2f is similar. Similarly, if all the arguments to a binary operator $\otimes$ are constant over each interval then the resulting values can be obtained by evaluation (CONSTARG). For example, suppose that for all $I_k \in \mathcal{I}$, the book-keeping map says that for all $x \in \widehat{I_k}$, the expression e_1 evaluates to k and e_2 evaluates to $k+1$. Then the book-keeping map for $e_1 +_{\mathsf{i}} e_2$ maps $x \in \widehat{I_k}$ to $2k+1$. Note that for bit-vector arithmetic operations, only the rule CONSTARG applies.

The rule for bitwise operations, when $\mathcal{B}$ is defined but the bits are not constant (BITOP), requires a refinement of two sets of intervals. This operation is defined as follows:

$$\mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \triangleq \{I_1 \cap I_2 \neq \varnothing : I_1 \in \mathcal{I}_1, I_2 \in \mathcal{I}_2\}$$

The refinement operation is necessary because the intervals over which $\mathcal{B}_1$ (or $\mathcal{A}_{1,\vec{\delta}}$) is defined piecewise can be different from those intervals for $\mathcal{B}_2$ (or $\mathcal{A}_{2,\vec{\delta}}$). For instance, for $\mathcal{I}_1 = \{[-3,0), (0,3]\}$ and $\mathcal{I}_2 = \{[-3,-1), (-1,1), (1,3]\}$, $\mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) = \{[-3,-1), (-1,0), (0,1), (1,3]\}$. Note that the refined intervals do not necessarily cover the original intervals: there can be inputs $x \in I_1$ where $I_1 \in \mathcal{I}_1$ and x is absent from all intervals of $\mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2)$. The shift operation also uses $\mathrm{refine}(\cdot,\cdot)$ and requires the shift amount to be a constant for each interval (SHIFT). These are in contrast to the rule CONSTARG that requires both of the arguments of a binary operator to be constant for each interval.

The symbolic representation is undefined for bit-level operations (BITOP and SHIFT). A floating-point operation results in an undefined book-keeping map and the symbolic representation is updated by applying the corresponding operation over real numbers and introducing a new error term δ' (FLOP)[4].

Remark. The rule FLOP describes an abstraction step: we are over-approximating the result obtained from a floating-point operator by introducing the δ variables that model rounding errors. Floating-point operations can be composed in non-trivial ways to generate exactly rounded results [8], whereas in the symbolic representations that we use, the rounding errors only increase with the number of operations. For example, there are no rounding errors if a floating-point number is multiplied by a power of two. However, the rule FLOP introduces a new δ variable for this operation.

The rounding operation requires an auxiliary function $\mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ (RND). This function splits the intervals in $\mathcal{I}$ further so that all floating-point numbers in each sub-interval round to the same integer:

$$\mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \triangleq \bigcup_{I \in \mathcal{I}} \{I_k \neq \varnothing : k \in \mathbb{Z}\}$$

where each $I_k \subset I$ is an interval such that $\mathcal{A}(I_k) \subset (k - \frac{1}{2}, k + \frac{1}{2})$, and $\mathcal{A}(I) \triangleq [\min_{x \in I, \vec{\delta}} \mathcal{A}_{\vec{\delta}}, \max_{x \in I, \vec{\delta}} \mathcal{A}_{\vec{\delta}}]$. For instance, for $\mathcal{I} = \{[-3,3]\}$ and $\mathcal{A}_{\vec{\delta}}(x) = (0.25 \times x)(1 + \delta_1)$, $\mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) = \{[-3, -\frac{2}{1-\epsilon}], [-\frac{2}{1+\epsilon}, \frac{2}{1+\epsilon}], [\frac{2}{1-\epsilon}, 3]\} = \mathcal{I}'$. The intervals created by $\mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ are not guaranteed to include all floating-point numbers that belong to the intervals of $\mathcal{I}$ (e.g., no interval of $\mathcal{I}'$ includes 2).

The rule SPLIT uses an auxiliary function $\mathrm{splitB}(\mathcal{I}, \mathcal{B})$ to split the intervals in $\mathcal{I}$ further so that $\mathcal{B}$ is constant on each sub-interval. For an interval I, we define

$$M(I, \mathcal{B}) \triangleq \max\{i : \forall x \in \{0,1\}^{64-i}.\, \exists n \in \{0,1\}^{64}. \\ \forall y \in \{0,1\}^i.\, (x,y) \in \widehat{I} \Rightarrow \mathcal{B}((x,y)) = n\}$$

Intuitively, $M(I, \mathcal{B})$ is the maximum bit position i such that for each choice of bits $x[63], \cdots, x[i]$ we have $\mathcal{B}(x[63], \cdots, x[0]) = n$ for some constant n regardless of the choice of bits $x[i-1], \cdots, x[0]$. By using $M(I, \mathcal{B})$, we define $\mathrm{splitB}(\mathcal{I}, \mathcal{B})$ as

$$\mathrm{splitB}(\mathcal{I}, \mathcal{B}) \triangleq \\ \bigcup_{I \in \mathcal{I}} \{[\mathrm{d2R}(l), \mathrm{d2R}(r)] \cap I \neq \varnothing : x \in \{0,1\}^{64-i}, \\ \text{where } i = M(I, \mathcal{B}),\ l = (x, 0^i), \text{ and } r = (x, 1^i)\}$$

Here, l and r are 64-bit vectors. For instance, for $\mathcal{B}(x) = x[63]01^9 0^{53}$, $I = [-3,3]$, and $\mathcal{I} = \{I\}$, we have $M(I, \mathcal{B}) = 63$ and $\mathrm{splitB}(\mathcal{I}, \mathcal{B}) = \{[-\infty, -0] \cap I, [+0, +\infty] \cap I\} = \{[-3,0], [0,3]\}$. Unlike $\mathrm{refine}(\cdot,\cdot)$ and $\mathrm{splitA}(\cdot,\cdot)$, the intervals created by $\mathrm{splitB}(\mathcal{I}, \mathcal{B})$ include all the floating-point numbers that belong to the intervals of $\mathcal{I}$. This rule is useful

[4] Modifying the rule FLOP from $\mathcal{A}' = (\mathcal{A}_1 \otimes \mathcal{A}_2)(1 + \delta')$ to $\mathcal{A}' = (\mathcal{A}_1 \otimes \mathcal{A}_2)(1 + \delta') + \delta''$ enables us to fully support denormals, where δ'' is a fresh variable with $|\delta''| < 2^{-1075}$.

to create intervals over which expressions evaluate to constant values, which are required for the rules LOOKUP, I2F, F2I$_{\text{CONST}}$, RND$_{\text{CONST}}$, CONSTARG and SHIFT.

If we mask out the lower order bits of the significand using a bitwise-and operation (BAND) then its affect on the symbolic representation can be modeled by an error term using the following result:

Lemma 1. *Let* $c \in \{0,1\}^{64}$ *and* $d = 1^{12+n}0^{52-n} \in \{0,1\}^{64}$ *for some* $n \in \mathbb{N}$. *Then* $\mathrm{d2R}(c \text{ AND } d) \in \{\mathrm{d2R}(c)(1+\delta') : |\delta'| < 2^{-n}\}$.

This result bounds the difference between the masked output and the input using error terms.

Our main result follows by induction on the derivation tree:

Theorem 1. *If* $e \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$, *then* $(\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$ *is consistent with* e, *and thus* $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ *is a symbolic abstraction of* e.

From the rules to construct symbolic abstractions, intervals in $\mathcal{I}$ cannot overlap with each other because each interval $I \in \mathcal{I}$ has the property that for all values in $\widehat{I}$, the value of some subexpression is guaranteed to be a constant, and because distinct intervals correspond to distinct constants. In constructing symbolic abstractions, the only step that requires manual intervention is the computation of $\mathrm{splitA}(\cdot,\cdot)$, which could be automated for our benchmarks in §5.

Next, we use this symbolic abstraction to bound the absolute error or the ULP error of an implementation e from its mathematical specification.

4.4 Computing Precision Loss

We describe a procedure to bound the precision loss of an implementation. Let e be an expression that implements a mathematical specification $f : \mathbb{R} \to \mathbb{R}$. The aim is to compute a bound Θ such that on any input the outputs of e and f differ by at most Θ. More formally,

Definition 5. $\Theta_a \in \mathbb{R}$ *is* a sound absolute error bound *for* e *and* f *over the interval* I *if for all* $x \in \widehat{I}$,

$$|\mathrm{d2R}(\mathcal{E}(e)(x)) - f(\mathrm{d2R}(x))| \leq \Theta_a$$

We have a similar definition for ULPs.

Definition 6. $\Theta_u \in \mathbb{R}$ *is* a sound ULP error bound *for* e *and* f *over the interval* I *if for all* $x \in \widehat{I}$,

$$\mathrm{ULP}(\mathcal{E}(e)(x), f(\mathrm{d2R}(x))) \leq \Theta_u$$

We first present a procedure to compute Θ_a. Let $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ be a symbolic abstraction of e. We compute

$$\Theta_1 = \max\left\{ \max_{x\in I,\vec{\delta}} \left| f(x) - \mathcal{A}_{\vec{\delta}}(x) \right| : I \in \mathcal{I} \right\} \quad (4)$$

This computation is an analytical optimization problem that can be solved by computer algebra systems. The time and memory required to solve the optimization problem increases with the number of variables. Moreover, $\mathcal{A}_{\vec{\delta}}(x)$ has many variables, because a new δ variable is created for each floating-point operation (the rule FLOP of Figure 5). Hence off-the-shelf solvers fail to solve Equation 4 as is.

For tractability, we simplify Equation 4. In our evaluation, the symbolic representations are polynomials with small degrees and the inputs are restricted to small ranges (§5). In this setting, we can prove that the terms in $\mathcal{A}_{\vec{\delta}}(x)$ that involve a product of multiple δ variables are negligible compared to other values. Informally, if the coefficients and input ranges of a polynomial are bounded by a constant c then the rounding error introduced by all δ terms with degree > 1 is bounded by $C = \frac{(4c)^n \epsilon}{1-\epsilon}$. The proof uses standard numerical analysis techniques and is omitted.

With this simplification, the expression $\left|f(x) - \mathcal{A}_{\vec{\delta}}(x)\right|$ can be rearranged such that $|f(x) - \mathcal{A}_{\vec{\delta}}(x)| \leq |M_0(x)| + \sum_i |M_i(x)||\delta_i|$, where $M_0(x)$ is C plus all the terms of $f(x) - \mathcal{A}_{\vec{\delta}}(x)$ having no δ, and $M_i(x)$ is the coefficient of δ_i in $f(x) - \mathcal{A}_{\vec{\delta}}(x)$. We use optimization techniques to maximize $|M_i(x)|$ for each interval $I \in \mathcal{I}$ (which is tractable because $M_i(x)$ is a function of a single variable) and report

$$\max_{x\in I} |M_0(x)| + \sum_i \left(\max_{x\in I} |M_i(x)| \right) \left(\max_{\delta_i} |\delta_i| \right) \quad (5)$$

as a bound on the absolute error over the interval I. The number of total optimization tasks required to obtain Θ_1 (Equation 4) is proportional to the product of the number of δ variables and the number of intervals in $\mathcal{I}$. In our evaluation, we observe that the number of optimization tasks is tractable and the time taken by each task is less than a second (§5).

Recall that X represents the interval of interest and we may have $\left(\bigcup_{I\in\mathcal{I}} I\right) \neq X$ due to the operations used in computing symbolic abstractions (i.e., $\mathrm{refine}(\cdot,\cdot)$ and $\mathrm{splitA}(\cdot,\cdot)$). Therefore, it is possible that $\Theta_a \neq \Theta_1$ if the input that results in the maximum absolute error belongs to the set $H = X \setminus \left(\bigcup_{I\in\mathcal{I}} I\right)$. So we compute

$$\Theta_2 = \max\left\{ |\mathrm{d2R}(\mathcal{E}(e)(x)) - f(\mathrm{d2R}(x))| : x \in \widehat{H} \right\} \quad (6)$$

In our experiments, $|\widehat{H}|$ is small enough that it is feasible to compute Θ_2 by brute force testing, i.e., by evaluating e and f on every floating-point number in $\widehat{H}$. Note that it is not feasible to perform brute force on $\widehat{X}$ as it contains an intractable number of floating-point numbers and thus symbolic abstractions are necessary. A sound absolute error bound is then given by $\Theta_a = \max\{\Theta_1, \Theta_2\}$.

Next, we compute a sound ULP error bound Θ_u. It can be computed from either a bound on absolute error or a bound on relative error (§3). The latter is the maximum of two quantities: the maximum relative error observed during testing on inputs in $\widehat{H}$ and the result of the following optimization problem:

$$\Theta_r = \max\left\{ \max_{x\in I,\vec{\delta}} \left| \frac{f(x) - \mathcal{A}_{\vec{\delta}}(x)}{f(x)} \right| : I \in \mathcal{I} \right\} \quad (7)$$

Rules for atomic expressions:

$$\frac{}{c \rhd (\{X\}, \mathrm{d2R}(c), c)}\ \text{CONST} \qquad \frac{}{\chi \rhd (\{X\}, \mathrm{Id}_{\mathbb{R}}, \mathrm{Id}_{\{0,1\}^{64}})}\ \text{INPUT}$$

Rules for operators (with all constant arguments):

$$\frac{\begin{array}{c} e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{constant} \\ \forall I \in \mathcal{I}.\ \mathcal{B}'\upharpoonright_{\widehat{I}} = \mathcal{E}(\mathsf{L}[\mathcal{B}\upharpoonright_{\widehat{I}}]) \wedge \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}'\upharpoonright_{\widehat{I}}) \end{array}}{\mathsf{L}[e] \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{LOOKUP} \qquad \frac{\begin{array}{c} e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{constant} \\ \forall I \in \mathcal{I}.\ \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}\upharpoonright_{\widehat{I}}) \end{array}}{\mathsf{i2f}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B})}\ \text{I2F}$$

$$\frac{\begin{array}{c} e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{constant} \\ \forall I \in \mathcal{I}.\ \mathcal{B}'\upharpoonright_{\widehat{I}} = \mathcal{E}(\mathsf{f2i}(\mathcal{B}\upharpoonright_{\widehat{I}})) \wedge \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}'\upharpoonright_{\widehat{I}}) \end{array}}{\mathsf{f2i}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{F2I}_{\text{CONST}} \qquad \frac{\begin{array}{c} e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{constant} \\ \forall I \in \mathcal{I}.\ \mathcal{B}'\upharpoonright_{\widehat{I}} = \mathcal{E}(\mathsf{round}(\mathcal{B}\upharpoonright_{\widehat{I}})) \wedge \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}'\upharpoonright_{\widehat{I}}) \end{array}}{\mathsf{round}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{RND}_{\text{CONST}}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \textsf{constant} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \textsf{constant} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \mathcal{B}'\upharpoonright_{\widehat{I}} = \mathcal{E}(\mathcal{B}_1\upharpoonright_{\widehat{I}} \otimes \mathcal{B}_2\upharpoonright_{\widehat{I}}) \wedge \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}'\upharpoonright_{\widehat{I}}) \end{array}}{e_1 \otimes e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{CONSTARG}$$

Rules for operators (with one or more non-constant argument(s)):

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \textsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \textsf{defined} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in \widehat{I}.\ \mathcal{B}'\upharpoonright_{\widehat{I}}(x) = \mathcal{B}_1\upharpoonright_{\widehat{I}}(x) \otimes_{\mathsf{b}} \mathcal{B}_2\upharpoonright_{\widehat{I}}(x) \end{array}}{e_1 \otimes_{\mathsf{b}} e_2 \rhd (\mathcal{I}', \perp, \mathcal{B}')}\ \text{BITOP}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \textsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \textsf{constant} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in \widehat{I}.\ \mathcal{B}'\upharpoonright_{\widehat{I}}(x) = \mathcal{B}_1\upharpoonright_{\widehat{I}}(x) \otimes_{\mathsf{s}} \mathcal{B}_2\upharpoonright_{\widehat{I}} \end{array}}{e_1 \otimes_{\mathsf{s}} e_2 \rhd (\mathcal{I}', \perp, \mathcal{B}')}\ \text{SHIFT}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}}, \_) \quad (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}})\ \textsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \mathcal{A}_{2,\vec{\delta}}, \_) \quad (\mathcal{I}_2, \mathcal{A}_{2,\vec{\delta}})\ \textsf{defined} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in I.\ \mathcal{A}'_{\vec{\delta}}\upharpoonright_I(x) = (\mathcal{A}_{1,\vec{\delta}}\upharpoonright_I(x) \otimes \mathcal{A}_{2,\vec{\delta}}\upharpoonright_I(x))(1+\delta'), \\ \text{where } \delta' \text{ is a fresh variable with a condition } |\delta'| < \epsilon \end{array}}{e_1 \otimes_{\mathsf{f}} e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \perp)}\ \text{FLOP}$$

$$\frac{\mathsf{round}(e) \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{defined}}{\mathsf{f2i}(e) \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})}\ \text{F2I} \qquad \frac{\begin{array}{c} e \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \_) \quad (\mathcal{I}, \mathcal{A}_{\vec{\delta}})\ \textsf{defined} \\ \mathcal{I}' = \mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \quad \forall I_k \in \mathcal{I}'.\ \mathcal{A}'_{\vec{\delta}}\upharpoonright_{I_k} = k \wedge \mathcal{B}'\upharpoonright_{\widehat{I_k}} = \widehat{k} \end{array}}{\mathsf{round}(e) \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{RND}$$

Rule to produce a book-keeping map that is constant on each interval:

$$\frac{\begin{array}{c} e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \textsf{defined} \\ \mathcal{I}' = \mathrm{splitB}(\mathcal{I}, \mathcal{B}) \quad \forall I \in \mathcal{I}'.\ \mathcal{A}'_{\vec{\delta}}\upharpoonright_I = \mathrm{d2R}(\mathcal{B}\upharpoonright_{\widehat{I}}) \end{array}}{e \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B})}\ \text{SPLIT}$$

Rule for masking:

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}}, \_) \quad (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}})\ \textsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \_\_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \textsf{constant} \\ \exists n \in \mathbb{N}.\ \forall I \in \mathcal{I}_2.\ \mathcal{B}_2\upharpoonright_{\widehat{I}} = 1^{12+n}0^{52-n}, \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in I.\ \mathcal{A}'_{\vec{\delta}}\upharpoonright_I(x) = (\mathcal{A}_{1,\vec{\delta}}\upharpoonright_I(x))(1+\delta'), \\ \text{where } \delta' \text{ is a fresh variable with a condition } |\delta'| < 2^{-n} \end{array}}{e_1\ \mathsf{AND}\ e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \perp)}\ \text{BAND}$$

Figure 5. The rules for constructing a symbolic abstraction.

f	LoC of `f`	LoC of `f`$_{\texttt{opt}}$	Reduction in size	Speedup
exp	38	28	10	1.6×
sin	66	42	24	1.5×
tan	107	89	18	1.1×
log	67	54	13	1.3×

Table 1. Important statistics of each implementation for case studies (LoC is lines of x86 assembly). The manually/automatically created variants have smaller number of instructions (column 4) and are faster (column 5) than their production counterparts.

In our evaluation, we compute bounds on the ULP error using both the relative and the absolute error and report the better bound.

5. Case Studies

We evaluate our technique on Intel's implementations of the positive difference function and three widely used transcendental functions: the sine function, the tangent function, and the natural logarithm function. We prove formal bounds on how much each implementation (`fdim`, `sin`, `tan`, and `log`) deviates from the corresponding mathematical specification ($\text{fdim}(\cdot,\cdot)$, $\sin(\cdot)$, $\tan(\cdot)$, and $\log(\cdot)$). These implementations are available as (undocumented) x86 binaries included in `libimf`, which is Intel's implementation of the C numerics library `math.h`. The library contains many different implementations for these functions that have differing performance on different processors and input ranges. We choose the implementations used in the benchmark set of [22] and perform the verification for these implementations.

Next, we manually modify these binaries to create implementations that have better performance at the cost of less precise results (Table 1). Using our technique, we are also able to prove formal bounds on the precision loss of these variants. The results are summarized in Table 2 and Figure 7.

We currently use Mathematica to solve Equation 5, though this is largely a matter of convenience. Mathematica provides functions to solve global optimization problems *analytically*, namely the `MaxValue`$[\cdot]$ and the `MinValue`$[\cdot]$ functions which find *exact* global optima of a given optimization problem (using several different algorithms, e.g., cylindrical algebraic decompositions).[5] The analytical optimization that we use is in contrast to typical *numerical* optimization that uses finite-precision arithmetic without providing formal guarantees. Other techniques, such as interval arithmetic or branch and bound optimization [23], can be used (instead of Mathematica) to solve Equation 5 soundly.

To compute Θ_2 in Equation 6, for each $x \in \widehat{H}$, we compare the floating-point result $\mathcal{E}(e)(x)$ computed by an evaluation of e on x, with the exact result $f(\text{d2R}(x))$ computed by Mathematica.

Even though Mathematica claims soundness guarantees, it does not produce a certificate of correctness with the solution. In the future we would like to replace Mathematica with a solver that produces machine-checkable certificates.

[5] `https://reference.wolfram.com/language/tutorial/ConstrainedOptimizationExact.html`

```
1  movapd %xmm0, %xmm2
2  cmpsd  $0x6,  %xmm1, %xmm0
3  andpd  %xmm0, %xmm1
4  andpd  %xmm2, %xmm0
5  subsd  %xmm1, %xmm0
6  retq
```

Figure 6. The x86 assembly code of `fdim`.

5.1 The `fdim` Implementation

As a warm up, consider the positive difference function of `math.h`: $\text{fdim}(x, y) \triangleq x - y$ if $x > y$ and 0 otherwise. Figure 6 shows Intel's implementation of `fdim`. It first sets each of the lower 64 bits of the register `xmm0` to be 0 if $x \le y$, and 1 otherwise (line 2). Next, it sets `xmm0` and `xmm1` to be both 0 if $x \le y$, and x and y otherwise (lines 3-4). The output is obtained by subtracting `xmm1` from `xmm0` (lines 5-6).

To obtain a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of `fdim`, we apply the technique described in §4.3 with $X = [-1, 1] \times [-1, 1]$. The instruction `cmpsd`[6] in line 2 results in the intervals $\mathcal{I} = \{I_1, I_2\}$, where $I_1 = \{(x, y) \in X : x \le y\}$ and $I_2 = \{(x, y) \in X : x > y\}$. The final symbolic representation is $\mathcal{A}_{\vec{\delta}}(x, y)\restriction_{I_1} = 0$ and $\mathcal{A}_{\vec{\delta}}(x, y)\restriction_{I_2} = (x - y)(1 + \delta_1)$. Computing a sound absolute error bound Θ_a for `fdim` and $\text{fdim}(\cdot,\cdot)$ over X is straightforward:

$$\Theta_a = \max_{(x,y)\in I_2,\ |\delta_1|<\epsilon} |(x - y)\delta_1| = 2\epsilon$$

The relative error bound (Equation 7) is ϵ so a sound ULP error bound is $\Theta_u = 2$.

5.2 The `sin` Implementation

Intel's `sin` implementation uses the following procedure to compute $\sin(x)$. It first computes an integer N and a reduced value d:

$$N = \text{round}\left(\frac{32}{\pi}x\right),\ d = x - \frac{\pi}{32}N \tag{8}$$

Then it uses the following trigonometric identity:

$$\sin(x) = \sin(d)\cos\left(\frac{\pi}{32}N\right) + \cos(d)\sin\left(\frac{\pi}{32}N\right)$$

[6] Due to space constraints, Figure 5 omits the rule for `cmpsd`.

	Interval	Θ_a	Θ_u	$\lvert\widehat{H}\rvert$	$\lvert\vec{\delta}\rvert$	$\lvert\mathcal{I}\rvert$
`fdim`	$[-1,1]\times[-1,1]$	2×10^{-16}	2	0	1	2
`exp`	$[-4,4]$	6×10^{-14}	14	36	29	13
`exp`$_{\texttt{opt}}$	$[-4,4]$	1×10^{-8}	2×10^{6}	36	19	13
`sin`	$\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$	2×10^{-16}	9	110	53	33
`sin`$_{\texttt{opt}}$	$\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$	2×10^{-11}	3×10^{5}	110	26	33
`tan`	$\left[0,\frac{17\pi}{64}\right)$	7×10^{-16}	12	89	84	10
	$\left[\frac{17\pi}{64},\frac{31\pi}{64}\right)^*$	2×10^{-13}	218	89	85	7
	$\left[\frac{31\pi}{64},\frac{\pi}{2}\right)^*$	2×10^{15}	9×10^{18}	89	85	1
`tan`$_{\texttt{opt}}$	$\left[0,\frac{17\pi}{64}\right)$	3×10^{-12}	3×10^{4}	89	69	10
	$\left[\frac{17\pi}{64},\frac{31\pi}{64}\right)^*$	4×10^{-10}	5×10^{5}	89	69	7
	$\left[\frac{31\pi}{64},\frac{\pi}{2}\right)^*$	2×10^{16}	9×10^{18}	89	69	1
`log`	$(0,4)\setminus\left[\frac{4095}{4096},1\right)$	8×10^{-14}	21	0	25	2^{21}
	$\left[\frac{4095}{4096},1\right)$	9×10^{-19}	1×10^{14}	0	25	1
`log`$_{\texttt{opt}}$	$(0,4)\setminus\left[\frac{4095}{4096},1\right)$	6×10^{-11}	5×10^{5}	0	12	2^{21}
	$\left[\frac{4095}{4096},1\right)$	1×10^{-18}	1×10^{14}	0	12	1

Table 2. Summary of results: For each implementation (column 1), for all inputs in the interval (column 2), Θ_a shows the bound on maximum absolute error and Θ_u shows the bound on maximum ULP error between the implementation and the exact mathematical result. The number of inputs that require testing ($|\widehat{H}|$), the number of δ variables ($|\vec{\delta}|$), and the number of intervals considered ($|\mathcal{I}|$) in the symbolic abstraction are also shown. The values of Θ_a and Θ_u on the rows with * need not to be sound.

The terms $\sin(d)$ and $\cos(d)$ are computed by a Taylor series expansion:

$$\sin(d) \approx d - \frac{d^3}{3!} + \frac{d^5}{5!} - \frac{d^7}{7!} + \frac{d^9}{9!}$$
$$\cos(d) \approx 1 - \frac{d^2}{2!} + \frac{d^4}{4!} - \frac{d^6}{6!} + \frac{d^8}{8!}$$

The Taylor series expansion includes an infinite number of terms. However, since d is small in magnitude, a small number of Taylor series terms shown above are sufficient to provide a good approximation of $\sin(d)$ and $\cos(d)$. The remaining terms, $\sin\left(\frac{\pi}{32}N\right)$ and $\cos\left(\frac{\pi}{32}N\right)$, are obtained by table lookups. A table in memory stores precomputed values $\sin\left(\frac{\pi}{32}i\right)$ and $\cos\left(\frac{\pi}{32}i\right)$ for each $i = 0,\cdots,63$ and the index $i = (N)$ AND `0x3f` is used to retrieve the correct value. The `sin` implementation uses bit-level operations to compute the index i and the final memory address for the lookup.

We modify `sin` to obtain a more efficient but less precise implementation `sin`$_{\texttt{opt}}$. These modifications include removing all subcomputations that have only a small effect on the final output. In particular, `sin`$_{\texttt{opt}}$ uses a Taylor series expansion of degree 5 (instead of 9). The *compensation* terms are also omitted. These are terms such as $(c_1 -_{\mathsf{f}} (c_1 +_{\mathsf{f}} c_2)) +_{\mathsf{f}} c_2$ that are 0 in the absence of rounding errors but are important for maintaining accuracy of floating-point computations (see the remark in §4.3). Moreover `sin`$_{\texttt{opt}}$ replaces some very small constants (e.g., 7.9×10^{-18}) by 0.

For $X = [-\frac{\pi}{2},\frac{\pi}{2}]$, we compute a symbolic abstraction $(\mathcal{I},\mathcal{A}_{\vec{\delta}})$ of `sin` by applying the technique described in §4.3. In the final abstraction $|\mathcal{I}| = 33$ and there are 53 δ's in $\mathcal{A}_{\vec{\delta}}$ (Table 2). The main step in the construction of a symbolic abstraction for `sin` (as well as `sin`$_{\texttt{opt}}$) is the application of the rule RND so that each of the resulting intervals contains inputs that map to the same N (Equation 8). A total of $|\widehat{H}| = 110$ inputs do not belong to any interval of $\mathcal{I}$, which is easily a small enough set that we can compute Θ_2 (Equation 6) via testing on each of these inputs.

We then use the procedure described in §4.4 to compute Θ_a and Θ_u over the interval X for `sin` and `sin`$_{\texttt{opt}}$ (Table 2 and Figure 7). Our main result for `sin` is a proof that for all inputs in $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$, the computed result differs from $\sin(x)$ only by 9 ULPs, that is, there are at most 9 floating-point numbers between the computed result and the mathematically exact result. For `sin`$_{\texttt{opt}}$, we have successfully traded a small loss in precision (bearable for many applications) for over 50% improvement in performance (Table 1).

5.3 The `tan` Implementation

Intel's `tan` implementation uses the following procedure to compute $\tan(x)$. To focus on the distinctive parts of the implementation, we assume that the input $x \in X = \left[0,\frac{\pi}{2}\right)$. The first step is to compute an integer N and a reduced value d:

$$N = \left\lfloor \frac{32}{\pi}x + \frac{1}{2} \right\rfloor,\ d = x - \frac{\pi}{32}N$$

Then we compute

$$\tan(x) \approx bR(x) + T_{15}(d),\ R(x) = \frac{1}{\frac{\pi}{2}-x} + \frac{c}{(\frac{\pi}{2}-x)^2}$$

Here, $b = 0$ if $x \in \left[0,\frac{17\pi}{64}\right)$ and $b = 1$ if $x \in \left[\frac{17\pi}{64},\frac{\pi}{2}\right)$, $c = 8.84372\cdots\times10^{-29}$, and $T_{15}(d) = \sum_{i=0}^{15} q_i d^i$ is a polynomial approximation of $\tan(x) - bR(x)$ of degree 15 where the polynomial coefficients depend on N. The coefficients $q_0,\cdots,q_{15}$ of $T_{15}(\cdot)$ are retrieved from a lookup table based on the index N. Note that, unlike `exp` and `sin`, `tan` includes rational terms $1/\left(\frac{\pi}{2}-x\right)$ and $1/\left(\frac{\pi}{2}-x\right)^2$ in order to minimize the precision loss near $x = \frac{\pi}{2}$.

The hand-optimized implementation `tan`$_{\texttt{opt}}$ uses a polynomial approximation of degree 12 (instead of 15) and omits an AND operation used by `tan` to mask out lower order bits of the significand of an intermediate result. We omit the compensation terms and obtain a total speed up of $1.1\times$ (Table 1).

To construct a symbolic abstraction of `tan` (and `tan`$_{\texttt{opt}}$), we apply the rules RND, SPLIT, and BAND. Again, we use Mathematica to compute an absolute error bound. However, the analytical optimization routines of Mathematica time out while computing Θ_1 in the presence of rational terms (i.e., if $x \geq \frac{17\pi}{64}$). Therefore, we use numerical optimization routines `NMaxValue`$[\cdot]$ and `NMinValue`$[\cdot]$ for the intervals that are subsets of $\left[\frac{17\pi}{64}, \frac{\pi}{2}\right)$. Since numerical optimization routines have no correctness guarantees, the computed error bounds for $x \geq \frac{17\pi}{64}$ need not to be sound. However, soundness is still maintained for $x < \frac{17\pi}{64}$ as these intervals are optimized analytically (using `MaxValue` and `MinValue`).

In Table 2, we observe that the ULP error over $\left[0, \frac{17\pi}{64}\right)$ is small (12 ULPs), and the ULP error over $\left[\frac{17\pi}{64}, \frac{31\pi}{64}\right)$ is slightly higher (218 ULPs). For the interval $J = \left[\frac{31\pi}{64}, \frac{\pi}{2}\right)$, we do not obtain good bounds. The absolute error is large on the interval J as the rational terms grow quickly near $\frac{\pi}{2}$. The relative error is large on J as one of the optimization task is the following:

$$\max_{x \in J} \left| \frac{1}{(\frac{\pi}{2} - x)^2 \tan(x)} \right|$$

For x close to $\frac{\pi}{2}$, the optimization objective is unbounded so the obtained bounds on relative error are large. Because neither the absolute error nor the relative error provides good bounds on ULP error, the bounds on ULP error are large for $x \in J$. The results for $\texttt{tan}_{\texttt{opt}}$ are similar (Table 2 and Figure 7).

5.4 The `log` Implementation

Intel's `log` implementation uses the following procedure to compute $\log(x)$ for $x > 0$. Let us denote the exponent of x by p and $f = f_1, f_2, \ldots, f_{52}$ denotes the bits of the significand of x. The implementation first constructs a single precision floating-point number $g = 1.f_1 \ldots f_{23}$. Next, the result $\frac{1}{g}$ obtained from a single precision reciprocal operation is converted to a double d'. Using $x = 1.f \times 2^p$, we have the following identity:

$$\begin{aligned} \log(x) &\approx \log(2^p) + \log(g) + \log(d' \times 1.f) \\ &\approx p\log(2) + \log\left(\frac{256}{i+128}\right) + \log(1+d), \end{aligned}$$

where $i = \text{round}(256d' - 128)$ and $d = d' \times 1.f - 1$. The quantity p is computed exactly by bit-level operations that extract the exponent of x. The quantity $\log\left(\frac{256}{i+128}\right)$ is computed by table lookups based on the index i. The lookup table stores the value $\log\left(\frac{256}{i+128}\right)$ for $i = 0, \cdots, 128$. Finally, since d is small in magnitude, $\log(1+d)$ is computed using a Taylor series of degree 7. The `log` implementation uses bit-level operations to compute p, g, and d'.

We hand-optimize `log` to create an implementation $\texttt{log}_{\texttt{opt}}$ that uses a Taylor series of degree 4 (instead of 7) and ignores some bit-manipulations of the significand. We also remove the compensation terms and obtain a total speedup of $1.3\times$ (Table 1).

To construct a symbolic abstraction of `log` (as well as $\texttt{log}_{\texttt{opt}}$), we apply the rule SPLIT to ensure that, for each interval, d' is a constant. Using $X = (0, 4)$, we obtain a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of `log` with $|\mathcal{I}| = 2^{21}$, $|\widehat{H}| = 0$, and 25 δ variables (Table 2). Unlike the other benchmarks, $|\widehat{H}| = 0$ as $\text{splitA}(\cdot, \cdot)$ is not used in constructing the symbolic abstraction. Since the number of intervals, $|\mathcal{I}|$, is large, we run 32 different instances of Mathematica in parallel. The optimizations are performed analytically and the obtained error bounds are sound. Moreover, each optimization task takes less than a second and the total wall clock time for `log` (resp. $\texttt{log}_{\texttt{opt}}$) was 16 hours (resp. 5 hours).

In our opinion, the analysis of the log function best illustrates the power of our technique: We are able to automatically reduce a very complex problem to millions of tractable subproblems, and the total time required compares favorably with the only alternative available today, which is an expert using an interactive theorem prover to construct the proof.

We present the most interesting results. Except for the interval $J = \left[\frac{4095}{4096}, 1\right)$, the ULP error is small: 21 ULPs (Table 2). For the interval J, the absolute error is very small: 9×10^{-19}. This small absolute error is expected as $\log(x)$ is close to zero on this interval ($\log(1) = 0$). However, due to the proximity to zero, even this small absolute error leads to a large ULP error (see the remark in §3). For `sin` and `tan`, we are able to get good bounds on ULP error near zero by using bounds on relative error. However, the relative error of `log` is large on the interval J as one of the optimization tasks is the following:

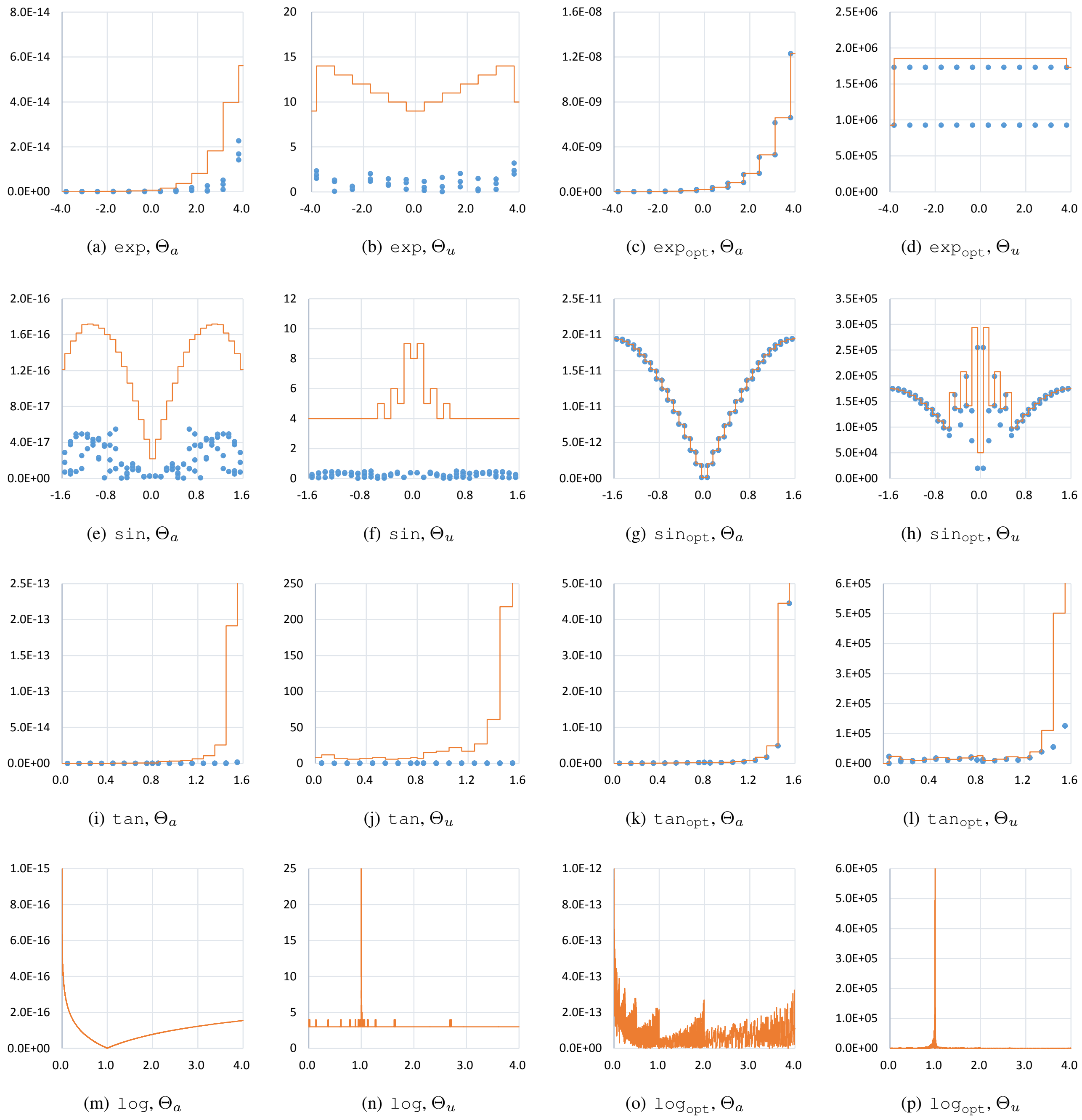
$$\max_{x \in J} \left| \frac{x - \frac{255}{256}}{\log x} \right|$$

For x close to one, the optimization objective is unbounded and the obtained bounds on relative error are large. Therefore, the bounds on ULP error are large for $x \in J$. The results for $\texttt{log}_{\texttt{opt}}$ are similar (Table 2 and Figure 7).

6. Prior Work

Due to their mathematical sophistication and practical importance, transcendental functions are used as benchmarks in many verification studies. However, prior studies have either focused on the verification of algorithms (and not implementations) or the verification of comparatively simple implementations that contain no bit-level operations.

In the absence of bit-level operations, a variety of techniques can be used to bound rounding errors: GAPPA uses interval arithmetic [6], FLUCTUAT uses affine arithmetic [7, 9], MATHSAT is an SMT solver that uses interval arithmetic for floating-point [10], and ROSA combines affine arithmetic with SMT solvers [5]. Interval arithmetic does not preserve dependencies between variables and affine arithmetic fits poorly with non-linearities. Hence, these approaches lead to


(a) `exp`, Θ_a (b) `exp`, Θ_u (c) $\texttt{exp}_{\texttt{opt}}$, Θ_a (d) $\texttt{exp}_{\texttt{opt}}$, Θ_u

(e) `sin`, Θ_a (f) `sin`, Θ_u (g) $\texttt{sin}_{\texttt{opt}}$, Θ_a (h) $\texttt{sin}_{\texttt{opt}}$, Θ_u

(i) `tan`, Θ_a (j) `tan`, Θ_u (k) $\texttt{tan}_{\texttt{opt}}$, Θ_a (l) $\texttt{tan}_{\texttt{opt}}$, Θ_u

(m) `log`, Θ_a (n) `log`, Θ_u (o) $\texttt{log}_{\texttt{opt}}$, Θ_a (p) $\texttt{log}_{\texttt{opt}}$, Θ_u

Figure 7. Each graph shows a bound on precision loss between an implementation and the mathematically exact result. For example, (a) plots a bound on absolute error between $\texttt{exp}(x)$ and e^x as a function of x. The brown lines represent the bounds obtained from symbolic abstractions and the blue dots signify the errors observed during explicit testing on inputs in $\widehat{H}$.

imprecise results (see the evaluation in [23]). Very recently optimization has been used in FPTAYLOR to bound rounding errors [23].

The problem that our technique solves is slightly different from the problems that previous methods do. FPTAYLOR bounds the difference between interpretations of an expression over the reals and over the floating-point numbers. Formally speaking, given a function $g : \mathbb{R} \rightarrow \mathbb{R}$, FPTAYLOR computes a bound on $|\mathrm{fp}(g)(x) - g(x)|$, where $\mathrm{fp}(g)$ is a function (from floating-point numbers to floating-point numbers) obtained from g by replacing all real-valued operations with the corresponding floating-point operations. In contrast, our technique bounds the difference between the polynomials obtained from symbolic abstractions of binaries and the true mathematical transcendentals. Due to the relationship between $\mathrm{fp}(g)$ and g, it is impossible to have that $\mathrm{fp}(g)$ is a polynomial *and* g is a transcendental, so FPTAYLOR and our technique solve different problems.

Similarly, GAPPA, FLUCTUAT, and ROSA also aim to bound the difference between two interpretations of the same expression: over the reals and over the floating-point numbers. Moreover, they do not encode transcendentals while our method requires encoding transcendentals.

Commercial tools such as FLUCTUAT and ASTRÉE [16] provide some support for mixed floating-point and bit-level code. These tools use abstract domains tailored to specific code patterns and reason soundly about low-level C implementations. In contrast, our approach is general and systematic. FLUCTUAT supports subdivision of input ranges, but its subdivision strategy is generic and requires no static analysis (e.g., repeatedly halving an input interval until the desired error bound on each subdivision is reached). The paper [15] subdivides according to the exponent bits to improve precision. Our work provides a general algorithm to construct these subdivisions; the need for this automatic interval construction to be sound leads to a distinctive characteristic of our subdivisions, the presence of floating-point numbers that are not covered by any interval.

Intel's processors contain instructions (e.g., `fsin`, `fcos`, etc.) that evaluate transcendental functions. The manual[7] for Pentium processors stated that a "rigorous mathematical analysis" had been performed and that the "worst case error is less than one ulp" for the output of these instructions. These claims were discovered to be incorrect and the instructions actually have large errors even for small arguments[8]. This incident indicates that obtaining accurate implementations of transcendental functions is non-trivial. Intel's latest manual[9] acknowledges the mistake ("the ulp error will grow above these thresholds") and recommends developers use the software implementations in Intel's Math Library. We are interested in verification of these implementations. A description of implementations that are similar to the ones that we verify can be found in [13].

Harrison describes machine-checkable proofs of algorithms that compute 64-bit $\sin(\cdot)$ [12] and 32-bit $\exp(\cdot)$ [11]. The verified algorithms are fairly close to the one employed by the implementations described in §5. Harrison's main result is a proof in HOL Light that the algorithms compute results within 0.57 ULPs of the mathematically exact result. The proof requires an extensive mathematical apparatus that seems necessary for such a precise bound. Harrison reports that the manual effort required for each such proof can vary from weeks to months [11].

In contrast, we have a general verification technique that can be readily applied to a variety of functions and their automatically or manually produced variants. This generality and better automation is achieved at the expense of analysis precision and the bounds we infer, while sharp enough to be useful, are loose compared to Harrison's manual proofs. Moreover, we have evaluated our technique only on small input ranges, whereas Harrison proves bounds for large ranges.

Techniques that provide statistical (as opposed to formal) guarantees include [17, 19, 22]. Analyses described in [1, 2, 4, 14] do not have any formal or statistical guarantees.

7. Discussion

Our technique is directly applicable to trading precision for performance. Most programs do not require all bits of precision and imprecise implementations can have better performance [21]. We believe that our verification technique significantly lowers the barrier to entry for developers who want to explore such trade-offs but are afraid of the subtleties associated with floating-point. The developers can create imprecise but efficient implementations either manually or by using tools such as [21, 22] and prove their correctness formally using our technique. Conversely, our technique can also be used to formally prove the correctness of transformations that improve precision [20]. These transformations are currently validated via sampling.

While we believe our approach is promising, there are important limitations that would be desirable to improve or remove altogether. Ideally, we would like to achieve bounds within 1 ULP. We have taken a step in this direction by demonstrating the first technique that can prove sound bounds on the ULP error for these implementations, which in some cases, are close to the desired result (e.g., 9 ULPs for `sin`). However, automatically proving that these routines are accurate to within 1 ULP requires further advances in verification techniques for floating-point that can address the imprecision introduced by the abstractions for rounding errors (see the remark in §4.3). Next, the inferred bounds are sound only for small ranges of inputs. Removing this restriction introduces new challenges: if the inputs belong

[7] Appendix G, `http://www.intel.com/design/pentium/MANUALS/24143004.pdf`

[8] `http://notabs.org/fpuaccuracy/`

[9] §8.3.1, `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

to a large range then the intermediate values could include NaNs or infinities. Moreover, the simplifications of the optimization problem described in §4.4 would no longer be sound. Orthogonally, our technique cannot handle all bit-level tricks. For example, a good approximation to the inverse square root of a floating-point number x is given by `0x5f3759df` $- (x \gg 1)$. During verification of this approximation using our technique, the rule SPLIT of Figure 5 creates an intractable number of intervals. Therefore, this task requires a different technique (e.g., exhaustive enumeration).

In general, the number of intervals can grow quickly with the number of arguments on which bit-level operations are performed. However, the functions in `math.h` have at most three arguments. Therefore, this situation does not arise in our particular application. We expect the number of intervals to be tractable even for multivariate implementations. E.g., the dyadic function `fdim` requires only two divisions (§5.1). Finally, since the optimization problems are independent, parallelization can improve scalability significantly (§5.4).

8. Conclusion

We have presented the first systematic method for verifying the behavior of binaries that mix floating-point and bit-level operations. We have also demonstrated that the technique is applicable directly to binaries corresponding to Intel's highly optimized implementations of transcendental functions.

Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also supported by NSF grants CCF-1160904 and CCF-1409813, a Samsung Fellowship, and a Microsoft Fellowship.

References

[1] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560, 2013.

[2] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, pages 453–462, 2012.

[3] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[4] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *PPOPP*, pages 43–52, 2014.

[5] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL*, pages 235–248, 2014.

[6] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):2:1–2:20, 2010.

[7] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, pages 53–69, 2009.

[8] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23 (1):5–48, 1991.

[9] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *FMICS*, pages 3–20, 2007.

[10] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140, 2012.

[11] J. Harrison. Floating-point verification in HOL Light: the exponential function. *Formal Methods in System Design*, 16 (3):271–305, 2000.

[12] J. Harrison. Formal verification of floating point trigonometric functions. In *FMCAD*, pages 217–233, 2000.

[13] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 4:234–251, 1999.

[14] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy - search-based floating point constraint solving for symbolic execution. In *ICTSS*, pages 142–157, 2010.

[15] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. Make it real: Effective floating-point reasoning via exact arithmetic. In *DATE*, pages 1–4, 2014.

[16] A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *ATx/WInG*, pages 55–70, 2012.

[17] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[18] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):12:1–12:41, 2008.

[19] G. C. Necula and S. Gulwani. Randomized algorithms for program analysis and verification. In *CAV*, page 1, 2005.

[20] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11, 2015.

[21] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC*, page 27, 2013.

[22] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs using tunable precision. In *PLDI*, pages 53–64, 2014.

[23] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*, pages 532–550, 2015.